Awk- An Advanced Filter

Introduction

awk is a programmable, pattern-matching, and processing tool available in UNIX. It works equally well with text and numbers. It derives its name from the first letter of the last name of its three authors namely Alfred V. Aho, Peter J. Weinberger and Brian W. Kernighan.

Simple awk Filtering

awk is not just a command, but a programming language too. In other words, awk utility is a pattern scanning and processing language. It searches one or more files to see if they contain lines that match specified patterns and then perform associated actions, such as writing the line to the standard output or incrementing a counter each time it finds a match.

Syntax:

awk option 'selection_criteria {action}' file(s)

Here, selection_criteria filters input and selects lines for the action component to act upon. The selection_criteria is enclosed within single quotes and the action within the curly braces. Both the selection_criteria and action forms an awk program.

Example: $ awk '/manager/ { print }' emp.lst

Output:

9876	Jai Sharma	Manager	Productions
2356	Rohit	Manager	Sales
5683	Rakesh	Manager	Marketing

In the above example, /manager/ is the selection_criteria which selects lines that are processed in the action section i.e. {print}. Since the print statement is used without any field specifiers, it prints the whole line.

Note: If no selection_criteria is used, then action applies to all lines of the file.

Since printing is the default action of awk, any one of the following three forms can be used:

awk '/manager/ ' emp.lst

awk '/manager/ { print }' emp.lst

awk '/manager/ { print $0}' emp.lst *$0 specifies complete line.*

Awk uses regular expression in sed style for pattern matching.

Example: awk –F "|" ' /r [ao]*/' emp.lst

Output:

2356	Rohit	Manager	Sales
5683	Rakesh	Manager	Marketing

Splitting a line into fields

Awk uses special parameter, $0, to indicate entire line. It also uses $1, $2, $3 to identify fields. These special parameters have to be specified in single quotes so that they will not be interpreted by the shell.

awk uses contiguous sequence of spaces and tabs as a single delimiter.

Example: awk –F "|" '/production/ { print $2, $3, $4 }' emp.lst

Output:

Jai Sharma |	Manager |	Productions
Rahul |	Accountant |	Productions
Rakesh |	Clerk |	Productions

In the above example, comma (,) is used to delimit field specifications to ensure that each field is separated from the other by a space so that the program produces a readable output.

Note: We can also specify the number of lines we want using the built-in variable NR as illustrated in the following example:

Example: awk –F "|" 'NR==2, NR==4 { print NR, $2, $3, $4 }' emp.lst

Output:

2	Jai Sharma	Manager	Productions
3	Rahul	Accountant	Productions
4	Rakesh	Clerk	Productions

printf: Formatting Output

The printf statement can be used with the awk to format the output. Awk accepts most of the formats used by the printf function of C.

Example: awk –F "|" '/[kK]u?[ar]/ { printf "%3d %-20s %-12s \n", NR, $2, $3}' >emp.lst

Output:

4	R Kumar	Manager
8	Sunil kumaar	Accountant
4	Anil Kummar	Clerk

Here, the name and designation have been printed in spaces 20 and 12 characters wide respectively.

Note: The printf requires \n to print a newline after each line.

Redirecting Standard Output:

The print and printf statements can be separately redirected with the > and | symbols. Any command or a filename that follows these redirection symbols should be enclosed within double quotes.

Example1: use of |

printf “%3d %-20s %-12s \n”, NR, $2, $3 | “sort”

Example 2: use of >

printf “%3d %-20s %-12s \n”, NR, $2, $3 > “newlist”

Variables and Expressions

Variables and expressions can be used with awk as used with any programming language. Here, expression consists of strings, numbers and variables combined by operators.

Example: (x+2)*y, x-15, x/y, etc..,

Note: awk does not have any data types and every expression is interpreted either as a string or a number. However awk has the ability to make conversions whenever required.

A variable is an identifier that references a value. To define a variable, you only have to name it and assign it a value. The name can only contain letters, digits, and underscores, and may not start with a digit. Case distinctions in variable names are important: Salary and salary are two different variables. awk allows the use of user-defined variables without declaring them i.e. variables are deemed to be declared when they are used for the first time itself.

Example: X= “4”
X= “3”
Print X
Print x

Note: 1. Variables are case sensitive.
2. If variables are not initialized by the user, then implicitly they are initialized to
zero.

Strings in awk are enclosed within double quotes and can contain any character. Awk strings can include escape sequence, octal values and even hex values.

Octal values are preceded by \ and hex values by \x. Strings that do not consist of numbers have a numeric value of 0.

Example 1: z = "Hello"
print z prints Hello

Example 2: y = “\t\t Hello \7”
print y prints two tabs followed by the string Hello and sounds a beep.

String concatenation can also be performed. Awk does not provide any operator for this, however strings can be concatenated by simply placing them side-by-side.

Example 1: z = "Hello" "World"
print z *prints* Hello World

Example 2 : p = “UNIX” ; q= “awk”
print p q *prints* UNIX awk

Example 3: x = “UNIX”
y = “LINUX”
print x “&” y *prints* UNIX & LINUX

A numeric and string value can also be concatenated.

Example : l = “8” ; m = 2 ; n = “Hello”
Print l m *prints 82 by converting m to string.*
Print l - m *prints 6 by converting l as number.*
Print m + n *prints 2 by converting n to numeric 0.*

Expressions also have true and false values associated with them. A nonempty string or any positive number has true value.

Example: if(c) *This is true if c is a nonempty string or positive number*.

The Comparison Operators

awk also provides the comparison operators like >, <, >=, <= ,==, !=, etc..,

Example 1 : $ awk –F “|” ‘$3 == “manager” || $3 == “chairman” {

> printf “%-20s %-12s %d\n”, $2, $3, $5}’ emp.lst

Output:

ganesh	chairman	15000
jai sharma	manager	9000
rohit	manager	8750
rakesh	manager	8500

The above command looks for two strings only in the third filed ($3). The second attempted only if (||) the first match fails.

Note: awk uses the || and && logical operators as in C and UNIX shell.

Example 2 : $ awk –F “|” ‘$3 != “manager” && $3 != “chairman” {
> printf “%-20s %-12s %d\n”, $2, $3, $5}’ emp.lst

Output:

Sunil kumaar	Accountant	7000
Anil Kummar	Clerk	6000
Rahul	Accountant	7000
Rakesh	Clerk	6000

The above example illustrates the use of != and && operators. Here all the employee records other than that of manager and chairman are displayed.

~ and !~ : The Regular Expression Operators:

In awk, special characters, called *regular expression operators* or *metacharacters*, can be used with regular expression to increase the power and versatility of regular expressions. To restrict a match to a specific field, two regular expression operators **~** (matches) and **!~** (does not match).

Example1: $2 ~ /[cC]ho[wu]dh?ury / || $2 ~ /sa[xk]s ?ena / *Matches second field*

Example2: $2 !~ /manager | chairman / *Neither manager nor chairman*

Note:

The operators ~ and !~ work only with field specifiers like $1, $2, etc.,.

For instance, to locate g.m s the following command does not display the expected output, because the word g.m. is embedded in d.g.m or c.g.m.

$ awk –F "|" '$3 ~ /g.m./ {printf "…..
prints fields including g.m like g.m, d.g.m and c.g.m

To avoid such unexpected output, awk provides two operators ^ and $ that indicates the beginning and end of the filed respectively. So the above command should be modified as follows:

$ awk –F "|" '$3 ~ /^g.m./ {printf "…..
prints fields including g.m only and not d.g.m or c.g.m

The following table depicts the comparison and regular expression matching operators.

Operator	Significance
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to
>=	Greater than or equal to
>	Greater than
~	Matches a regular expression
!~	Doesn't matches a regular expression

Table 1: Comparison and regular expression matching operators.

Number Comparison:

Awk has the ability to handle numbers (integer and floating type). Relational test or comparisons can also be performed on them.

Example: $ awk –F "|" '$5 > 7500 {
> printf "%-20s %-12s %d\n", $2, $3, $5}' emp.lst

Output:

ganesh	chairman	15000
jai sharma	manager	9000
rohit	manager	8750
rakesh	manager	8500

In the above example, the details of employees getting salary greater than 7500 are displayed.

Regular expressions can also be combined with numeric comparison.

Example: $ awk –F "|" '$5 > 7500 || $6 ~/1980$/' {
> printf "%-20s %-12s %d\n", $2, $3, $5, $6}' emp.lst

Output:

ganesh	chairman	15000	30/12/1950
jai sharma	manager	9000	01/01/1980
rohit	manager	8750	10/05/1975
rakesh	manager	8500	20/05/1975
Rahul	Accountant	6000	01/10/1980
Anil	Clerk	5000	20/05/1980

In the above example, the details of employees getting salary greater than 7500 or whose year of birth is 1980 are displayed.

Number Processing

Numeric computations can be performed in awk using the arithmetic operators like +, -, /, *, % (modulus). One of the main feature of awk w.r.t. number processing is that it can handle even decimal numbers, which is not possible in shell.

Example: $ awk –F "|" '$3' == "manager" {

> printf "%-20s %-12s %d\n", $2, $3, $5, $5*0.4}' emp.lst

Output:

jai sharma	manager	9000	3600
rohit	manager	8750	3500
rakesh	manager	8500	3250

In the above example, DA is calculated as 40% of basic pay.

Variables

Awk allows the user to use variables of there choice. You can now print a serial number, using the variable kount, and apply it those directors drawing a salary exceeding 6700:

```
$ awk –F"|"  '$3 == "director" && $6 > 6700 {
    kount =kount+1
    printf " %3f  %20s %-12s %d\n", kount,$2,$3,$6 }' empn.lst
```

The initial value of kount was 0 (by default). That's why the first line is correctly assigned the number 1. awk also accepts the C- style incrementing forms:

```
Kount ++
Kount +=2
Printf "%3d\n", ++kount
```

THE –f OPTION: STORING awk PROGRAMS INA FILE

You should holds large awk programs in separate file and provide them with the .awk extension for easier identification. Let's first store the previous program in the file empawk.awk:

```
$ cat empawk.awk
```

Observe that this time we haven't used quotes to enclose the awk program. You can now use awk with the –f *filename* option to obtain the same output:

```
Awk –F"|" –f empawk.awk empn.lst
```

THE BEGIN AND END SECTIONS

Awk statements are usully applied to all lines selected by the address, and if there are no addresses, then they are applied to every line of input. But, if you have to print something before processing the first line, for example, a heading, then the BEGIN section can be used gainfully. Similarly, the end section useful in printing some totals after processing is over.

The BEGIN and END sections are optional and take the form

```
BEGIN {action}
END {action}
```

These two sections, when present, are delimited by the body of the awk program. You can use them to print a suitable heading at the beginning and the average salary at the end. Store this program, in a separate file **empawk2.awk**

Like the shell, awk also uses the # for providing comments. The BEGIN section prints a suitable heading , offset by two tabs (\t\t), while the END section prints the average pay (tot/kount) for the selected lines. To execute this program, use the –f option:

```
$awk –F"|" –f empawk2.awk empn.lst
```

Like all filters, awk reads standard input when the filename is omitted. We can make awk behave like a simple scripting language by doing all work in the BEGIN section. This is how you perform floating point arithmetic:

```
$ awk 'BEGIN {printf "%f\n", 22/7 }'
3.142857
```

This is something that you can't do with **expr.** Depending on the version of the awk the prompt may be or may not be returned, which means that awk may still be reading standard input. Use [*ctrl-d*] to return the prompt.

BUILT-IN VARIABLES

Awk has several built-in variables. They are all assigned automatically, though it is also possible for a user to reassign some of them. You have already used NR, which signifies the record number of the current line. We'll now have a brief look at some of the other variable.

The FS Variable: as stated elsewhere, awk ues a contiguous string of spaces as the default field delimeter. FS redefines this field separator, which in the sample database happens to be the |. When used at all, it must occur in the BEGIN section so that the body of the program knows its value before it starts processing:

```
BEGIN {FS="|"}
```

This is an alternative to the –F option which does the same thing.

The OFS Variable: when you used the print statement with comma-separated arguments, each argument was separated from the other by a space. This is awk's default output field separator, and can reassigned using the variable OFS in the BEGIN section:

```
BEGIN { OFS="~" }
```

When you reassign this variable with a ~ (tilde), awk will use this character for delimiting the print arguments. This is a useful variable for creating lines with delimited fields.

The NF variable: NF comes in quite handy for cleaning up a database of lines that don't contain the right number of fields. By using it on a file, say emp.lst, you can locate those lines not having 6 fields, and which have crept in due to faulty data entry:

```
$awk 'BEGIN { FS = "|" }
➢  NF !=6 {
```

- Print “Record No “, NR, “has ”, “fields”}’ empx.lst

The FILENAME Variable: FILENAME stores the name of the current file being processed. Like grep and sed, awk can also handle multiple filenames in the command line. By default, awk doesn’t print the filename, but you can instruct it to do so:

‘$6<4000 {print FILENAME, $0 }’

With FILENAME, you can device logic that does different things depending on the file that is processed.

ARRAYS

An array is also a variable except that this variable can store a set of values or elements. Each element is accessed by a subscript called the **index. Awk** arrays are different from the ones used in other programming languages in many respects:

- They are not formally defined. An array is considered declared the moment it is used.
- Array elements are initialized to zero or an empty string unless initialized explicitly.
- Arrays expand automatically.
- The index can be virtually any thing: it can even be a string.

In the program empawk3.awk, we use arrays to store the totals of the basic pay, da, hra and gross pay of the sales and marketing people. Assume that the da is 25%, and hra 50% of basic pay. Use the tot[] array to store the totals of each element of pay, and also the gross pay:
Note that this time we didn’t match the pattern sales and marketing specifically in a field. We could afford to do that because the patterns occur only in the fourth field, and there’s no scope here for ambiguity. When you run the program, it outputs the average of the two elements of pay:

$ awk –f empawk3.awk empn.lst

C-programmers will find the syntax quite comfortable to work with except that awk simplifies a number of things that require explicit specifications in C. there are no type declarations, no initialization and no statement terminators.

Associative arrays

Even though we used integers as subscripts in the tot [ ] array, awk doesn’t treat array indexes as integers. Awk arrays are associative, where information is held as *key-value* pairs. The index is the key that is saved internally as a string. When we set an array element using mon[1]=”mon”, awk

converts the number 1 to a string. There's no specified order in which the array elements are stored. As the following example suggests, the index "1" is different from "01":

```
$ awk 'BEGIN {
    direction ["N"] = "North" ; direction ["S"] ;
    direction ["E"] = "East" ; direction ["W"] = "West" ;]
    printf("N is %s and W is %s \n", direction["N"], direction ["W"]);

    Mon[1] = "Jan"; mon["1"] = "january" ; mon["01"] = "JAN" ;
    Printf("mon is %s\n", mon[1]);
    Printf("mon[01] is also %s\n",mon[01]);
    Printf("mon[\"1\"] is also %s \n", mon["1"]);
    Printf("But mon[\"01\"] is %s\n", mon["01"]);
   }
```

There are two important things to be learned from this output. First, the setting with index "1" overwrites the setng made with index 1. accessing an array element with subscript 1 and 01 actually locates the element with subscript "1". Also note that mon["1"] is different from mon["01"].

ENVIRON[]: The Environment Array:

You may sometimes need to know the name of the user running the program or the home directing awk maintains the associative array, ENVIRON[], to store all environment variables. This POSIX requirement is met by recent version of awk including *nawk* and *gawk*. This is how we access the shell variable , HOME and PATH, from inside an awk program:

```
$nawk 'BEGIN {
>print "HOME" "=" ENVIRON["HOME"]
>print "PATH" "=" ENVIRON["PATH"]
>}'
```

FUNCTIONS

Awk has several built in functions, performing both arithmetic and string operations. The arguments are passed to a function in C-style, delimited by commas and enclosed by a matched pair of parentheses. Even though awk

allows use of functions with and without parentheses (like printf and printf()), POSIX discourages use of functions without parentheses.

Some of these functions take a variable number of arguments, and one (length) uses no arguments as a variant form. The functions are adequately explained here so u can confidently use them in perl which often uses identical syntaxes.

There are two arithmetic functions which a programmer will except **awk** to offer. **int** calculates the integral portion of a number (without rounding off),while sqrt calculates square root of a number. **awk** also has some of the common string handling function you can hope to find in any language. There are:

length: it determines the length of its arguments, and if no argument is present, the enire line is assumed to be the argument. You can use length (without any argument) to locate lines whose length exceeds 1024 characters:

awk –F"|" 'length > 1024' empn.lst

you can use length with a field as well. The following program selects those people who have short names:

awk –F"|" 'length ($2) < 11' empn.lst

index(*s1, s2*): it determines the position of a string s2within a larger string s1. This function is especially useful in validating single character fields. If a field takes the values a, b, c, d or e you can use this function n to find out whether this single character field can be located within a string abcde:

x = index ("abcde", "b")
This returns the value 2.

substr (stg, m, n): it extracts a substring from a string *stg. m* represents the starting point of extraction and n indicates the number of characters to be extracted. Because string values can also be used for computation, the returned string from this function can be used to select those born between 1946 and 1951:

awk –F"|" 'substr($5, 7, 2) > 45 && substr($5, 7, 2) < 52' empn.lst
2365|barun sengupta|director|personel|11/05/47|7800|2365
3564|sudhir ararwal|executive|personnel|06/07/47|7500|2365
4290|jaynth Choudhury|executive|production|07/09/50|6000|9876
9876|jai sharma|director|production|12/03/50|7000|9876

you can never get this output with either **sed** *and* ***grep*** because regular expressions can never match the numbers between 46 and 51. Note that awk does indeed posses a mechanism of identifying the type of expression from its context. It identified the date field string for using substr and then converted it to a number for making a numeric comparison.

split(stg, arr, ch): it breaks up a string stg on the delimiter ch and stores the fields in an array arr[]. Here's how yo can convert the date field to the format YYYYMMDD:

```
$awk –F "|" '{split($5, ar, "/"); print "19"ar[3]ar[2]ar[1]}' empn.lst
19521212
19501203
19431904
……..
```

You can also do it with **sed,** but this method is superior because it explicitly picks up the fifth field, whereas **sed** would transorm the only date field that it finds.

system: you may want to print the system date at the beging of the report. For running a UNIX command within a awk, you'll have to use the system function. Here are two examples:

```
BEGIN {
        system("tput clear")        Clears the screen
        system("date")              Executes the UNIX date command
        }
```

CONTROL FLOW- THE if STATEMENT:

Awk has practically all the features of a modern programming language. It has conditional structures (the if statement) and loops (while or for). They all execute a body of statements depending on the success or failure of the *control command.* This is simply a condition that is specified in the first line of the construct.

Function	***Description***
int(x)	returns the integer value of x
sqrt(x)	returns the square root of x
length	returns the complete length of line
length(x)	returns length of x
substr(stg, m, n)	returns portion of string of length n, starting from position m in string stg.
index(1s, s2)	returns position of string s2 in string s1
splicit(stg, arr, ch)	splicit string stg into array *arr* using ch as delimiter, returns number of fields.
System("cmd")	runs UNIX command cmd and returns its exit status

The **if** statement can be used when the && and || are found to be inadequate for certain tasks. Its behavior is well known to all programmers. The statement here takes the form:

```
If  (condition is true) {
        Statement
} else {
Statement
}
```

Like in C, none of the control flow constructs need to use curly braces if there's only one *statement* to be executed. But when there are multiple actions take, the statement must be enclosed within a pair of curly braces. Moreover, the control command must be enclosed in parentheses.

Most of the addresses that have been used so far reflect the logic normally used in the **if** statement. In a previous example, you have selected lines where the basic pay exceeded 7500, by using the condition as the selection criteria:

```
$6 > 7500 {
```

An alternative form of this logic places the condition inside the action component rather than the selection criteria. But this form requires the if statement:

```
Awk –F "|" '{ if ($6 > 7500) printf …….
```

if can be used with the comparison operators and the special symbols ~ and !~ to match a regular expression. When used in combination with the logical operators || and &&, awk programming becomes quite easy and powerful. Some of the earlier pattern matching expressions are rephrased in the following, this time in the form used by **if:**

```
if ( NR > = 3 && NR <= 6 )
if ( $3 == "director" || $3 ==  "chairman" )
if ( $3 ~ /^g.m/ )
if ( $2 !~ / [aA]gg?[ar]+wal/ )
if ( $2 ~[cC]ho[wu]dh?ury|sa[xk]s?ena/ )
```

To illustrate the use of the optional **else** statement, let's assume that the dearness allowance is 25% of basic pay when the latter is less than 600, and 1000 otherwise. The **if-else** structure that implants this logic looks like this:

```
If ( $6 < 6000 )
        da = 0.25*$6
else
        da = 1000
```

You can even replace the above **if** construct with a compact conditional structure:

```
$6 <  6000 ? da = 0.25*$6 : da = 1000
```

This is the form that C and **perl** use to implement the logic of simple **if-else** construct. The ? and : act as separator of the two actions.

When you have more than one statement to be executed, they must be bounded by a pair of curly braces (as in C). For example, if the factors determining the hra and da are in turn dependent on the basic pay itself, then you need to use terminators:

```
If ( $6 < 6000 ) {
        hra = 0.50*$6
```

```
        da = 0.25*$6
}else {
        hra = 0.40*$6
        da = 1000
}
```

LOOPING WITH for:

a**wk** supports two loops – **for** and **while.** They both execute the loop body as long as the control command returns a true value. **For** has two forms. The easier one resembles its C counterpart. A simple example illustrates the first form:

```
for (k=0; k<=9; k+=2)
```

This form also consists of three components; the first component initializes the value of k, the second checks the condition with every iteration, while the third sets the increment used for every iteration. **for** is useful for centering text, and the following examples uses **awk** with **echo** in a pipeline to do that:

```
$echo "
>Income statement\nfor\nthe month of august, 2002\nDepartment : Sales" |
>awk ' { for (k=1 ; k < (55 –length($0)) /2 ; k++)
>printf "%s"," "
>printf $0}'

Income statement
        for
the month of August, 2002
    Department : Sales
```

The loop here uses the first **printf** statement to print the required number of spaces (page width assumed to be 55). The line is then printed with the second **printf** statement, which falls outside the loop. This is useful routine which can be used to center some titles that normally appear at the beginning of a report.

Using for with an Associative Array:

The second form of the **for** loop exploits the associative feature of **awk**'s arrays. This form is also seen in perl but not in the commonly used languages like C and java. The loop selects each indexof an array:

```
for ( k in array )
        commamds
```

Here, k is the subscript of the array *arr*. Because k can also be a string, we can use this loop to print all environment variables. We simply have to pick up each subscript of the ENVIRON array:

```
$ nawk 'BIGIN {
>for ( key in ENVIRON )
>print key "=" ENVIRON [key]
>}'

LOGNAME=praveen
MAIL=/var/mail/Praveen
PATH=/usr/bin::/usr/local/bin::/usr/ccs/bin
TERM=xterm
HOME=/home/praveen
SHELL=/bin/bash
```

Because the index is actually a string, we can use any field as index. We can even use elements of the array counters. Using our sample databases, we can display the count of the employees, grouped according to the disgnation (the third field). You can use the string value of $3 as the subscript of the array kount[]:

```
$awk –F'|' '{ kount[$3]++ }
>END { for ( desig in kount)
>print desig, kount[desig] }' empn.lst

g.m         4
chairman    1
executive   2
director    4
manager     2
d.g.m       2
```

The program here analyzes the databases to break up of the employees, grouped on their designation. The array kount[] takes as its subscript non-numeric values g.m., chairman, executive, etc.. **for** is invoked in the END section to print the subscript (desig) and the number of occurrence of the subscript (kount[desig]). Note that you don't need to sort the input file to print the report!

LOOPING WITH while

The **while** loop has a similar role to play; it repeatedly iterates the loop until the command succeeds. For example, the previous **for** loop used for centering text can be easily replaced with a **while** construct:

```
k=0
while (k < (55 – length($0))/2) {
        printf "%s"," "
        k++
}
print $0
```

The loop here prints a space and increments the value of k with every iteration. The condition (k < (55 – length($0))/2) is tested at the beginning of every iteration, and the loop body only if the test succeeds. In this way, entire line is filled with a string spacesbefore the text is printed with **print $0.**

Not that the **legth** function has been used with an argument ($0). This **awk** understands to be the entire line. Since length, in the absence of arguments, uses the entire line anyway, $0 can be omitted. Similarly, **print $0** may also be replaced by simply **print.**

Programs

1)awk script to delete duplicate

lines in a file.

```
BEGIN { i=1;}
{
 flag=1;
  for(j=1; j<i && flag ; j++ )
  {
        if( x[j] == $0 )
        flag=0;
  }
if(flag)
  {
        x[i]=$0;
        printf "%s \n",x[i];
        i++;
  }
}
```

Run1:

```
[root@localhost shellprgms]$ cat >for7.txt
hello
world
world
hello
this
```

is

this

Output:

```
[root@localhost shellprgms]$ awk -F "|" -f 11.awk for7.txt
hello
world
this
is
```

2)awk script to print the transpose of a matrix.

```
 BEGIN{
          system(“tput clear”)
          count =0
             }
{
                      split($0,a);
  for(j=1;j<=NF;j++)

{  count = count+1
   arr[count] =a[j]
 }
    K=NF
}
END{
printf(“Transpose\n”);
for(j=1;j<=K;j++)
{
   for(i=j; i<=count; i=i+K)
{
                        printf(“%s\t”, arr[i]);
                    }
```

```
                printf("\n");
        }
 }
```

Run1:

```
 [root@localhost shellprgms]$ qwk –f p8.awk
        2       3
        5       6
```

Transpose

```
        2       5
        3       6
```

3)Awk script that folds long line into 40 columns. Thus any line that exceeds 40 Characters must be broken after 40th and is to be continued with the residue. The inputs to be supplied through a text file created by the user.

```
BEGIN{
start=1; }
{ len=length;
for(i=$0; length(i)>40; len-=40)
{
print substr(i,1,40) "\\"
i=substr(i,41,len);
}
print i; }
```

Run1:

```
[root@localhost shellprgms]$ awk -F "|" -f 15.awk sample.txt
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\
aaaaaaaaaaa
aaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\
```

aaaaaaaaa

Output:

[root@localhost shellprgms]$ cat sample.txt

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

aaaaaaaaaaaa

aaaaaaaaaaaaaa

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

4)This is an awk program to provide extra spaces at the end of the line so that the line length is maintained as 127.

```
awk ‘ { y=127 – length($0)
        printf “%s”, $0
        if(y > 0)
        for(i=0;i<y;i++)
           printf “%s”, “ ”
        printf “\n”
       }’ foo
```

5)A file contains a fixed number of fields in the form of space-delimited numbers. This is an awk program to print the lines as well as total of its rows.

```
awk ‘{ split($0,a)
       for (i=1;i<=NF;i++) {
        row[NR]+=a[$i]
        }


     printf “%s”, $0
     printf “%d\n”, row[NR]
    } ’ foo
```
